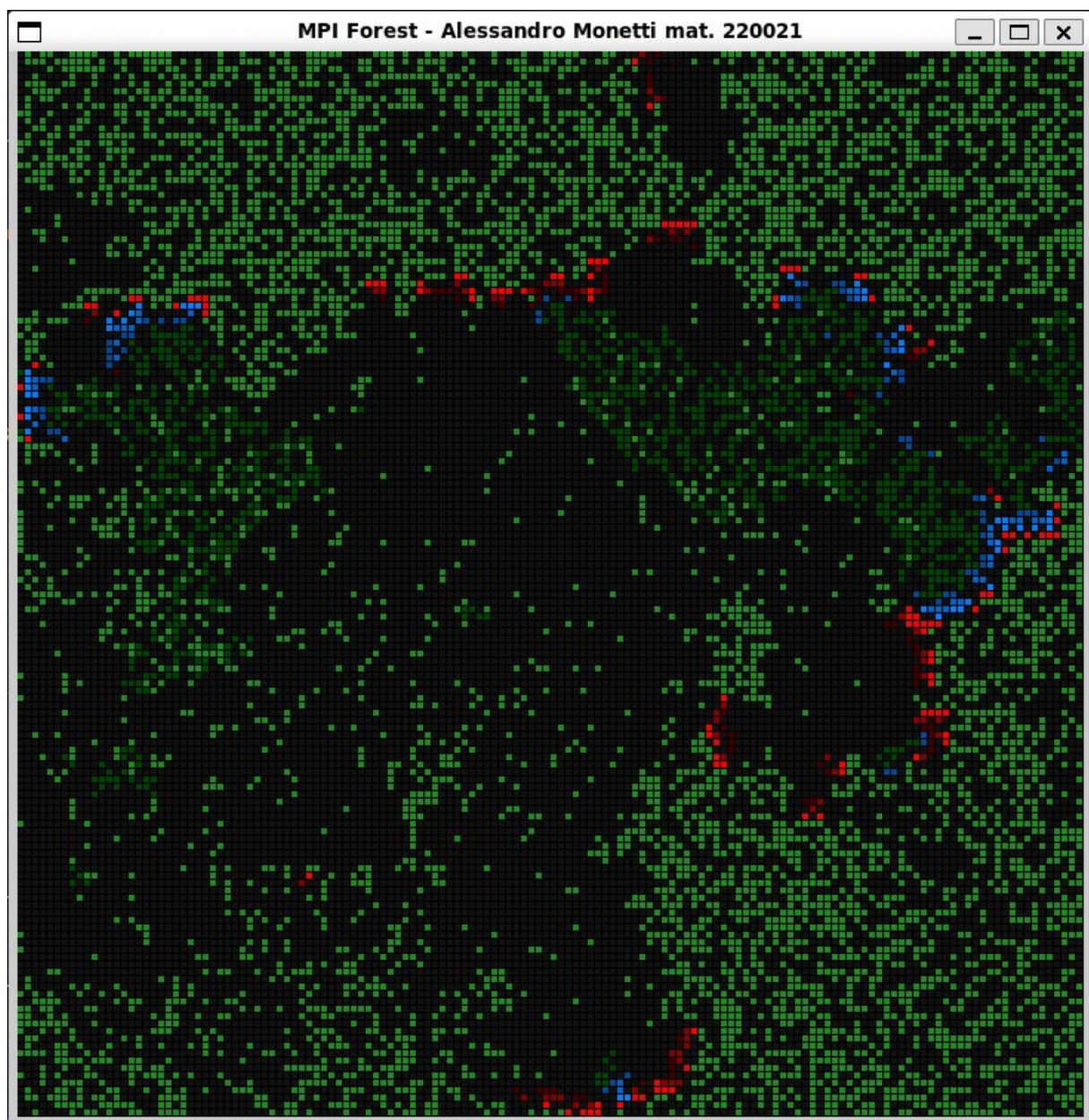


Progetto di
Algoritmi Paralleli e Sistemi Distribuiti

Forest Fire+



Alessandro Monetti - mat. 220021

L'automa cellulare “Forest Fire”

Il modello originale

Forest Fire è un'automa cellulare che simula il bioma della foresta e il suo peggior nemico: il fuoco.

All'istante 0 la foresta risulta completamente spoglia, e passo dopo passo, le celle che la compongono iniziano a trasformarsi in alberi, su una base puramente probabilistica.

Ogni albero, in qualsiasi momento, può prendere fuoco. Ciò accade se quest'ultimo viene colpito da un fulmine o uno dei suoi vicini è già in fiamme.

Una volta bruciato, al passo successivo l'albero scompare lasciando al suo posto una cella vuota.

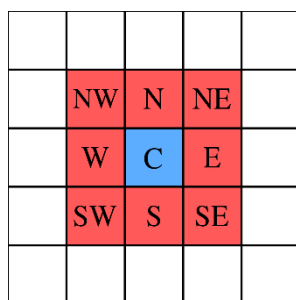


Fig. 1: Il modello di vicinato utilizzato è quello di “Moore” – courtesy of en.wikipedia.org

Il modello rivisitato

Il modello presentato, in versione appositamente rivisitata, presenta alcune sottili differenze dalla versione originale.

Ogni cella può avere **9** differenti **stati**, raccolti nel sorgente in una semplice enumerazione così composta:

```
states {EMPTY, TREE, BURNT_TREE, BURNING_LOW, BURNING_MID, BURNING_HIGH, WATER_LOW, WATER_MID, WATER_HIGH};  
        0         1         2         3         4         5         6         7         8
```

Come è possibile notare dalla dichiarazione di cui sopra, è stata aggiunta l'**acqua**, che può essere gettata casualmente sul fuoco per poi seguirlo e spegnerlo, lasciando al suo posto un albero bruciacchiato, che ha più possibilità di una cella vuota di diventare albero.

Sono stati aggiunti, inoltre, sia per fuoco che per acqua, 2 stati ulteriori ciascuno, utilizzati per permettere l'effetto grafico di “sfumatura”.

I cambiamenti di stato

Di seguito la guida sui possibili cambiamenti di stato:

EMPTY: Può rimanere **EMPTY** oppure diventare **TREE** (con una probabilità dello 0,2%);

TREE: Può rimanere **TREE** oppure diventare **BURNING_HIGH** (tramite vicinanza a **BURNING_HIGH** o colpito da un fulmine con una probabilità dello 0,002%);

BURNT_TREE: Può rimanere **BURNT_TREE** oppure diventare **TREE** (con una probabilità dell' 1%);

BURNING_LOW: Può diventare **EMPTY** oppure diventare **WATER_LOW** se entra a contatto di: { **WATER_LOW**, **WATER_MID**, **WATER_HIGH** };

BURNING_MID: Può diventare **BURNING_LOW** oppure diventare **WATER_MID** se entra a contatto di: { **WATER_MID**, **WATER_HIGH** };

BURNING_HIGH: Può diventare **BURNING_MID** oppure diventare **WATER_HIGH** (tramite vicinanza a **WATER_HIGH** o bagnato dagli aerei antincendio con una probabilità dello 0,02%)

WATER_LOW: Diventa **BURNT_TREE**;

WATER_MID: Diventa **WATER_LOW**;

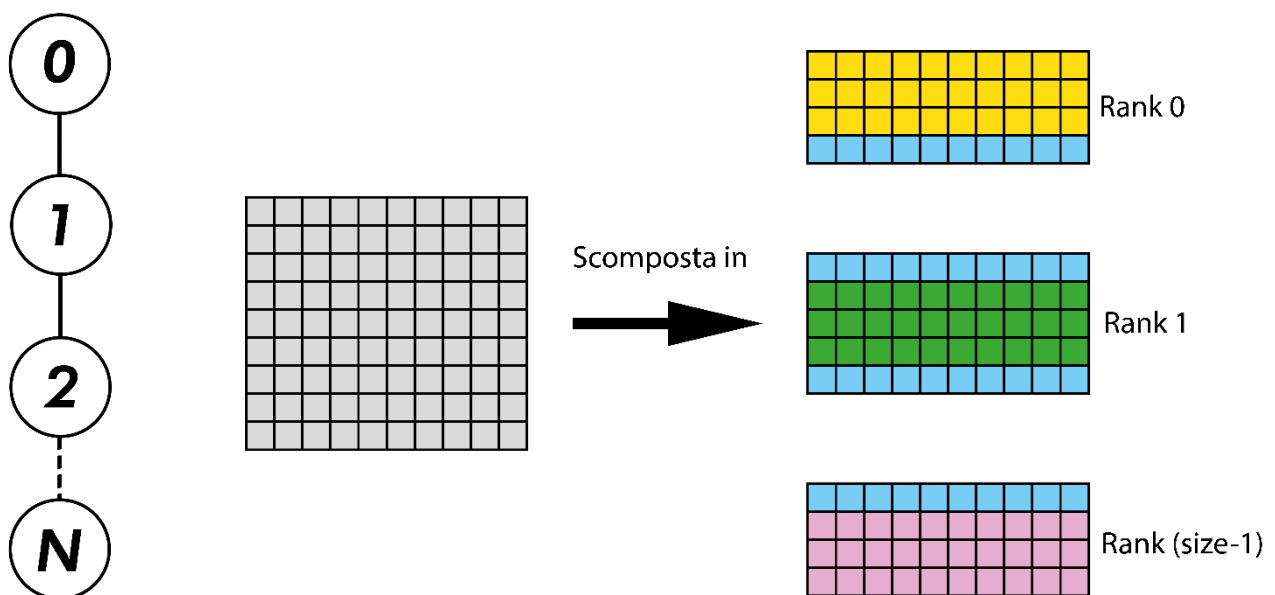
WATER_HIGH: Diventa **WATER_MID**;

L'implementazione

Per lo sviluppo della versione parallela di questo progetto si è deciso di affiancare al comune linguaggio di programmazione “C” il protocollo di comunicazione “**MPI**” (*Message Passing Interface*), usato generalmente per l'elaborazione parallela su cluster di computer.

Per poter rendere possibile ciò è stato necessario dividere i dati da gestire ai vari **rank** (*processi*).

Per la suddivisione dei processi si è pensato di fare utilizzo di una topologia virtuale di tipo cartesiano monodimensionale (senza periodicità, poiché non necessaria) integrata in MPI (**MPI_Cart_create()** ed **MPI_Cart_shift()**), con orientamento verticale (puramente frutto di convenzione), per ottenere un alto grado di parallelizzazione riducendo l'**overhead** (tempo di esecuzione extra impiegato oltre alla computazione, largamente influenzato dalle comunicazioni).



Figg. 2-3: A sinistra la topologia virtuale utilizzata, a destra una demo di scomposizione della matrice, con ghost cells (in azzurro)

Ad ogni iterazione del ciclo principale, ogni rank effettuerà le seguenti operazioni nell'ordine:

- Invio bordi (**non bloccante**) ai nodi adiacenti;
- Applicazione **funzione transizione** all'interno del nodo eccetto bordi;
- Ricezione bordi (**bloccante**) dai nodi adiacenti e inserimento in ghost cells;
- Applicazione funzione transizione ai bordi del nodo;
- Scambio piani di lettura e scrittura;
- *Invio matrice lettura al nodo che si occupa della **grafica** **
- *Aggiornamento status variabile stop da rank 0 **

**Necessari esclusivamente per impiego grafica*

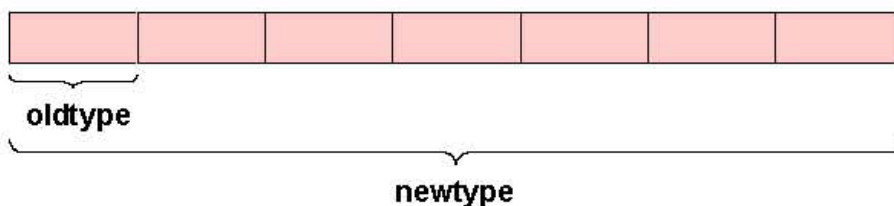
Le comunicazioni vengono fatte tra i processi utilizzando le funzioni **MPI_Isend()** ed **MPI_Recv()**.

La simulazione prevede anche una visualizzazione grafica utilizzando la libreria "**allegro5**" per rappresentare lo stato corrente dell'automa.

Per rendere più agevole l'invio dei dati tra i vari rank si è inoltre scelto di fare uso di due **MPI_Datatype** (*contigui*) derivati creati ad hoc:

- **MPI_MYROW** composto da nCols **MPI_INT**
- **MPI_MYLOCALMATRIX** composto da (nRows/size)*nCols **MPI_INT**

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



- C:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fig. 4: *MPI_Type_contiguous(...)* – courtesy of Prof. William Spataro

I risultati

Premesse

Per vedere in gioco l'effettiva utilità del calcolo parallelo, sono state prese diverse misurazioni, sulla seguente macchina:

CPU: Ryzen 7 3700X @ 4,35GHz (8 Cores/16 Threads)

RAM: Crucial Ballistix 4x8 (32GB) @ 3600MHz (Cas Latency di 16)

I tempi (mostrati di seguito in *millisecondi*) sono stati presi su una media di 10 misurazioni per ciascun test case (250 misurazioni totali) e sono stati presi (ad eccezione dell'ultimo gruppo) dopo 100 step, con 1, 2, 4, 8 e 16 processi, e matrici di dimensioni 128x128, 256x256, 512x512 e 992x992, senza flag di ottimizzazione in fase di compilazione.

```
After 100 steps: {
  CA matrix size (128 x 128)
    1 thread : 25,271247      Speedup: 1      Efficiency: 1
    2 threads: 12,784562     Speedup: 1,976700 Efficiency: 0,988350
    4 threads: 6,654327      Speedup: 3,797716 Efficiency: 0,949429
    8 threads: 4,204924      Speedup: 6,009917 Efficiency: 0,751239
    16 threads: 2,973017     Speedup: 8,500202 Efficiency: 0,531262

  CA matrix size (256 x 256)
    1 thread : 101,078122    Speedup: 1      Efficiency: 1
    2 threads: 51,106359     Speedup: 1,977799 Efficiency: 0,988899
    4 threads: 25,735392     Speedup: 3,927592 Efficiency: 0,981898
    8 threads: 18,733431     Speedup: 5,395601 Efficiency: 0,674450
    16 threads: 14,682482    Speedup: 6,884266 Efficiency: 0,430266

  CA matrix size (512 x 512)
    1 thread : 406,222751    Speedup: 1      Efficiency: 1
    2 threads: 205,633711    Speedup: 1,975467 Efficiency: 0,987733
    4 threads: 109,032439    Speedup: 3,725705 Efficiency: 0,931426
    8 threads: 72,203691     Speedup: 5,626066 Efficiency: 0,703258
    16 threads: 41,620500    Speedup: 9,760160 Efficiency: 0,610010

  CA matrix size (992 x 992)
    1 thread : 1527,508490   Speedup: 1      Efficiency: 1
    2 threads: 775,265994    Speedup: 1,970302 Efficiency: 0,985151
    4 threads: 415,489074    Speedup: 3,676410 Efficiency: 0,919102
    8 threads: 260,839269    Speedup: 5,856129 Efficiency: 0,666005
    16 threads: 149,464295   Speedup: 10,219888 Efficiency: 0,638743
}

After 1000 steps: {
  CA matrix size (992 x 992)
    1 thread : 20792,490355  Speedup: 1      Efficiency: 1
    2 threads: 10716,475906  Speedup: 1,940235 Efficiency: 0,970117
    4 threads: 5750,790332   Speedup: 3,615588 Efficiency: 0,903897
    8 threads: 3424,068797   Speedup: 6,072451 Efficiency: 0,759056
    16 threads: 1980,622202  Speedup: 10,497959 Efficiency: 0,656122
}
```

Fig. 5: Le misurazioni dei tempi

Considerazioni

Nota: La grafica (e la funzione **MPI_Gather()** utilizzata per comunicare al **rank** che si occupa di quest'ultima) è stata scorporata dal calcolo dei tempi mostrati in figura, in quanto altererebbe di molto le misurazioni.

Essa è stata utilizzata solo a scopo dimostrativo circa l'effettivo funzionamento dell'automa,

Dando uno sguardo all'immagine precedente, possiamo notare come lo **speedup** (ovvero il rapporto fra il tempo impiegato in maniera sequenziale e il tempo impiegato in parallelo) cresca all'aumentare del numero di processi impiegati; ciò nonostante, l'**efficienza** rimane lontana dalla migliore auspicabile (ovvero un rapporto tra speedup e numero di processi uguale ad 1)

Per completezza e curiosità sono state anche effettuate le stesse misurazioni CON l'impiego della **MPI_Gather()**, ed è stato notato come quest'ultima introduca una tangibile quantità di **overhead**, soprattutto per quanto riguarda 8 e 16 processi.

I risultati sono visibili nel file di testo "*times.txt*" allegato alla presente relazione.